A High Concurrency HTTP Server Architecture

Chris Pauley

April 2019

Abstract

This paper discusses the architecture behind an HTTP server used focused on asynchronously handling network IO while maintaining cache locality by pinning requests to threads. The architecture benefits from concurrency from both asynchronous operations and multithreading. Additional performance improvements are achieved through load balancing and proper memory management for fewer cache misses and remote memeory access.

1. Introduction

There are two popular architectures for HTTP servers: One that utilizes one thread per connection, and others that utilize non-blocking event systems (commonly known as asynchronous). This paper explores the problems solved by developing a structure that utilizes pieces from both of these architectures and learns from some of their downsides.

The problems addressed by this paper are as follows:

- Optimizing the event system and thread pool to operate together.
- Better cache utilization through strict requirements on localization.
- The changing landscape of advancements in processors and how to architect to benefit from them.
- Load Balancing work queues while maintaining requirements for concurrency.

Finally, the paper will analyze the performance results of this architecture and compare it across the architectures it borrows from.

This architecture is novel as a whole but borrows from existing models. The event-loop is very old as well as thread-per-connection models. This architecture marries the two in a novel way. The key points describing this architecture are as follows

- **Multiple requests per thread.** As opposed to a *thread-per-connection* model, this server handles multiple requests on a single thread concurrently.
- **Single thread per request.** Each request will have all work processed on a single thread.
- **Event based request handling.** Each thread has an event loop handling request work asynchronously.

2. Server Architecture

The main thread creates a pool of work threads, each thread containing a queue and a number of workers. The main thread then loops over accept. When a connection is accepted, a connection object is created from the socket and queued to the thread with the lowest load factor. The work threads run on an asynchronous timer loop until there is a connection. When the connection is accepted the thread assigns the connection to a worker which handles the asynchronous reading, processing, and writing to the request.

Each thread has its own asynchronous context, and thus can have multiple workers handling requests concurrently. Each have a unique context to keep each request tied to a specific thread which reduces the number of cache misses. Additionally this provides an opportunity for higher level load balancing which is specifically beneficial to the load expected by this server.

Due to the expected load being heavily biased toward network communication (either to/from the client or to/from the database server) an asynchronous event loop allows for many more connections to be handled than a simple thread per connection setup. As the server accepts many more connections it also creates many more connections to the database server, moving the performance bottleneck to the database itself. With a properly scaled database server (or cluster) and well written handlers this will balance the architecture providing very straightforward scaling options.

2.1 Connection Flow

The main thread accepts all connections. When a connection is made, the main thread wraps the socket in a container class which records timings, logs connection errors, and ensures the socket is closed properly. The time at which the connection was accepted is recorded.

The handling thread will then pass the connection to an available worker. This worker handles all asynchronous operations. It will read the connection and parse the HTTP request. Once the request is parsed, an appropriate

handler is picked based on the HTTP verb and target. If no such match is found, the connection is marked as errored and skips to writing the error to the connecting socket. For certain handlers additional steps are taken to authenticate the request.

The handler is then called and given request and response objects. Another timing is taken from the time the handler starts until it returns. If any exceptions are thrown during the execution of the handler they are caught, logged, and an error response is written to the socket. When the handler returns the response object is serialized, cleanup is done on the handler, the response is written to the socket. Another timing is taken and is compared to the first timing for the connection to determine the total duration of the connection. This is then used to update the load lookup table. The connection object is destructed, closing the connection and the worker is reset.

2.2 Concurrency And Cache Locality

A significant portion of the requests will require network communication by the server. This work could be waiting on files from an archive server or database communication. This communication presents a certain amount of time the server must wait. Concurrency is then necessary in order to handle multiple connections in a timely manner.

The server takes advantage of two forms of concurrency: Multithreading and asynchronicity. Multithreading is standard in and performance focused webserver as it allows two or more requests to be handled in parallel. Asynchronicity, however, provides the server with an opportunity to take advantage of the wait time produced by network communication.

The standard way to manage a multi-threaded server is to have a pool of threads, each taking a single request at a time. This removes the need for shared memory within the request handlers and simplifies handler code. However, when the requests handlers wait for any reason (network traffic, for example) the thread blocks and is completely idle.

The standard way to manage a multithreaded asynchronous server is to have a pool of threads each polling on an event loop and handling individual pieces of a request as the work is dispatched. If there are requests to be handled no thread in the queue will be idle (unless a blocking call is made within some unit of work). A request may be read on one thread, processed on another, and written on yet another thread.

Moving a request from thread to thread removes the ability to properly utilize CPU cache. When processing very low-overhead items such as HTTP requests there are significant performance improvements from improving cache locality. This becomes increasingly important as multiple sockets become involved. If a request moves between a thread on one socket to another the request memory goes from being local to remote and must be accessed through slower means such as the processor interconnect. The same problem exists on a smaller scale within single processor machines: after switching threads the memory being accessed misses on L1 after being switched, and could subsiquently miss all the way to main memory.

Modern Processor Advancements

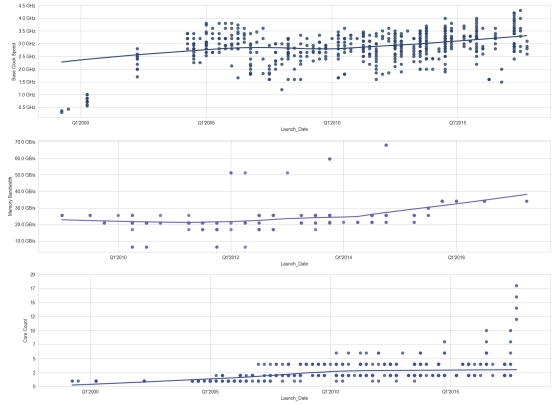

Figure 1: Processor Advancements

As displayed in Figure 1, over the past several years processor clock speed has not increased dramatically however core count has increased dramatically. Additionally, memory bandwidth has not increased at the rate that total processor speed[1] has. In order to take advantage of modern processor improvements applications must better utilize multithreaded architectures and minimize remote memory access.

There are two ways to reduce remote memory access: Reduce the amount of memory being used, and increase the percentage of the memory being used in cache. Managing the L1 and L2 cache such that the only threads accessing the cache are the near threads reduces the cache size and minimizes the number of times memory must be read before getting a hit. To do this on the server, the memory being accessed by a thread needs to be consistent.

For these reasons the strict alignment of requests to threads result in performance improvements even on a single processor, but these improvements scale with new hardware. As more cores are added to single sockets and more sockets become available as hardware scales this architecture will scale accordingly.

[1] *base clock speed * hardware thread count*

2.3 Load Balancing

In order to both utilize a thread pool and pin requests to a single thread, it is necessary to balance the load across threads. Each thread contains a queue of connections to be handled, but the load that connection brings is not known until the connection has been accepted and read (and is pinned to a thread).

In order to determine the load represented by a connection, a table is created mapping types of connections to average processing time. Every time a request is completed, the table is updated taking the new timing into account. When a request is read it checks the table for a matching connection type and updates the work thread with the load expectency. Combining the load expectency for each connection being handled on the thread results in an expected duration to complete all open connections.

The load lookup table must be specific enough to map requests with distinct processing times, but not so specific that many requests do not have a match. In the case of this server the matching key is a combination of the target path and HTTP verb. It may be necessary in the future to also consider certain query parameters or headers.

When a connection is accepted each work thread's load value is examined and the thread with the lowest value is assigned the connection. It is possible for a thread to be assigned a connection which may take significantly longer than other connections while already having a significantly sized queue relative to other threads.

For this reason each thread is capable of stealing work from other threads much in the same way as the linux thread scheduler. However, as opposed to the thread scheduler, this work stealing does not upset cache locality at all. The memory in the queues is shared and once the connection is dequeued, the memory associated with that connection stays on the thread that dequeued it.

3. Performance

A baseline test was performed to test the server against the C10K problem presented by Dan Kegel[1]. The test consisted of simultaneously executing 20,000 requests which return 4kb of data from disk. The test was completed with a median request time of 30ms and none exceeding 300ms.

Another test was developed to test the real-world use case of the server. The intention of this test is to determine expected latencies for a given load. The load used for the test is 1,000 concurrent users running the 4kb request, a request to load information from the database, and a request sending data to the server to be saved to the database. In total this results in 30,000 requests.

Authors Note: The graphs and timings provided in this paper were the result of tests run on hardware that is not optimized for this use case and are purely for comparitive purposes between models. The CPU running the server is able to show the advantages of multi-threading due to having 12 cores and 24 threads, however is bottlenecked by the database server which is running on outdated consumer grade hardware not originally purposed for this use case.

3.1 Total Request Time

The test was run several times to examine the improvements of utilizing different forms of concurrency. The configurations consisted of a single-threaded multi-worker test, a multi-threaded single-worker test, and a multi-threaded multi-worker test. The number of threads and number of workers were kept consistent when multiple were used.

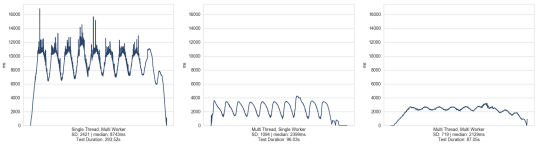

Figure 2: Total Request Time

In Figure 2, the latencies show a significant performance boost between the single and multi-threaded tests, but somewhat similar latencies between multi-threaded tests (note the difference in y-axis scale for the first result). This shows there is very little time spent waiting on asio events, and as such there is not much room for performance improvement in terms of medin response time. However, the total request times for the multi-thread multi-worker model are far more consistent and predictable.

The multi-thread single-worker results show a very volatile pattern of request timings. This is due to each of the threads blocking simultaneously creating congestion for a period of time. This can also be seen in Figure 3 where the lack of available threads causes significant increases in the time to accept a connection. These situations are mitigated by the multi-thread multi-worker model by accepting a connection during the time spent waiting on sockets to become ready. The more opportunities for work to be performed by the thread, the less congestion will occur.

3.2 Load Modelling

For the multi-thread multi-worker model predicting how long a request might take given the current state of the

server is a fairly simple equation. This provides the necessary variables for predicting necessary hardware to handle a given amount of users.

$$\frac{queue\ size * duration * concurrency}{median\ handler\ time}$$

Where *concurrency* is the total number of workers across all threads and *queue size* is the number of connections that can wait in the queue prior to being handled by a worker. This equation provides a total number of requests that can be concurrently requested and processed within the provided duration. This provides an index for a given set of hardware and load profile which is invaluable when planning for hardware scaling. It is also a helpful index to measure the performance impact an individual handler might have on the server's performance as a whole.

3.3 Further Improvements

Further investigation provides important insight into the potential benefit of running asynchronous work: Faster connect times and more predictable latencies.

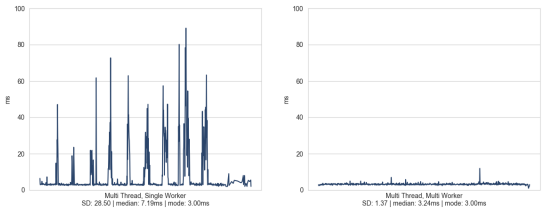

Figure 3: Time to Connect

When all threads are simultaneously blocking the next request cannot connect until one of the connections is completed. The spikes in figure 3 show the requests waiting to connect as all threads are busy. In the multi-thread multi-worker model these circumstances are avoided as there are more opportunities for a request to be accepted. This also reduces the median request time as the spikes are removed, but the statistical mode remains the same.

The improvement between accept times in these two models show an opportunity for further improvement by converting the request processing from being a blocking operation to an asynchronous one. This will allow operations on the database to be handled asynchronously, which will provide opportunities for work to be completed when the thread would be idle otherwise. Because these happen within the handlers themselves, it should reduce the overall median request time and increase the total load possible to be handled by a given set of hardware.

Removing blocking code from the request handlers should also reduce areas prone to congestion. As a long request is waiting on some connection (for example a database query) the thread will still handle other connections in the queue. This prevents any thread from initiating a multiple connections then proceeding to wait on a single request to finish before returning, which covers the area of requests not protected by the work stealing mechanism.

4. Scaling

The primary focus of this architecture is to scale across cores. As new CPUs with more cores become available this server should scale in kind. This is important as over the past several years CPUs have not advanced as much in terms of base clock speed as in core count. There are several important considerations with growing core counts, one of which is caching and memory access.

While more cores are becoming available, memory bandwidth has not increased substantially. As a result in terms of this server more requests are able to be handled concurrently, but bandwidth per request reduces as concurrency scales. This makes utilizing the CPU cache even more important as CPU cache is still fast to access at scale. For these same reasons this server scales well in multi-socket and NUMA scenarios as requests are tied to their threads. The ways in which this problem is addressed by the server are mentioned in detail in Section 1.2.

The area that will need to be addressed by this server as it scales is scaling `accept()`. Currently, a blocking accept call is made in a loop on the main thread. While this works well up to a significant scale, it will eventually need to be modified. Likely what will need to happen is to have accept on an event loop utilizing `epoll` or each thread will run accept with the `SO_REUSEPORT` and address the issues with dropped connections[2] using `SO_INCOMING_CPU` or `SO_ATTACH_REUSEPORT_{E/C}BPF`.

Another factor in scaling the server is to reduce the amount of time spent in the handler. The amount of time spent in the handler directly affects the number of concurrent users can behandled without experiencing timeouts. This can be done by reducing the amount of processing done by the handlers or by reducing the amount of time the handlers spend waiting on other machines. In the tests run for this paper the latencies could be dramatically reduced by improving the hardware running on the database server.

5. Conclusion

The performance metrics show that this architecture solves some important problems of an HTTP server at scale. However, it still can be improved upon and may not be a suitable option for most cases. One aspect not explored in this paper is the complexity involved in this architecture comes with a requirement of certain expertise when working with the core structure.

The findings in developing this architecture could impact the way some architectures are approached especially in regards to concurrency models. The requirment of restricting a request to a single thread may be difficult to implement but with the changes in processor architectures seen in the past few years it is increasingly necessary. The available improvements discussed in this paper could also improve the performance of this server further.

The source for the data and charts in this table are available along with the source of the paper at github.com/chris-pauley/http-server-concurrency-paper.

6. References

[1] Kegel, Dan. The c10k problem.. Available from: http://www.kegel.com/c10k.html

[2] wahern, LWN User. Available from: https://lwn.net/Articles/542866/

[3] Drepper, Ulrich. What every programmer should know about memory, part 6.. Available from: https://lwn.net/Articles/256433/